# Federated Consistency in Geographically Distributed Systems

Benjamin Bengfort and Pete Keleher
Department of Computer Science
University of Maryland, College Park, MD, USA
{bengfort,keleher}@cs.umd.edu

***Abstract*—Consistency in a distributed storage system is usually thought of as one of a few discrete categories: eventual, causal, sequential, or linearizable and distributed systems are designed with homogenous replica types to fit one of those categories. In this paper we present Federated Consistency, a heterogenous system model that allows individual replica servers to select their own consistency level, specifying a quality of service at a particular location while still providing global guarantees. This model, applied to variable latency geographic systems that are partition prone allow for flexibility in consistency that can adapt in response to the local network environment.**

## I. Introduction

The rise of on-demand computing resources and the Cloud have made distributed systems the default approach to scaling applications for many users in a variety of geographic locations. In particular, data replication is used to increase availability, throughput, durability, and fault tolerance by ensuring that objects can be accessed on multiple servers as locally as possible. Although some coordination is necessary to ensure that replication happens correctly, many systems favor a relaxation in consistency in order to meet the performance requirements of modern, mobile applications. This is partially because the application layer can define its own mechanisms for handling differently consistent behavior but it is primarily because such systems are implemented in data center contexts that enjoy stable, low-latency connections which allow optimistic approaches and provide consistent views of data *most of the time* [8], [9].

We might, therefore, generally categorize most modern distributed storage systems and NoSQL databases as not having strong consistency and using some consensus algorithm for control and synchronization when necessary. As a result consistency is usually described in a discrete, data-centric fashion: weak or strong; eventual, causal, or sequential and no longer described in client-centric terms [10]. As replication becomes more prevalent, however, it is not enough to simply lay the burden of conflict at the feet of clients and there has been recent interest in instead redefining consistency along a spectrum whose dimensions are the strictness of ordering writes and the potential staleness of reads [37], [26], [1], [3], [21].

By defining consistency in terms of ordering and staleness it is easy to see that the root cause of the tradeoff between performance and correctness is message latency, where a common case is when messages do not or cannot arrive due to node failure or network partitions. It has been noted that message latency is the key factor in determining "how consistent" a system is either due to staleness in eventually consistent systems [7] or by preventing progress in sequential consistency systems implemented with consensus [18]. The advent of distributed storage as a service has allowed systems to adapt consistency at runtime by taking advantage of a stable network environment [12], [13], [20] and has shifted the focus away from replication in weakly-connected, dynamic, or mobile networks even though it is the environment that has the primary role in determining the behavior of replication and potential guarantees [33], [11]. [PJK: Important...]We believe that local, user-oriented distributed systems should augment cloud services rather than be replaced by them; and in some cases, such as disaster recovery or search and rescue, may be the only available system.

In this paper we present a novel approach to flexible consistency via the federation of a heterogenous system of replica servers that implement a variety of consistency protocols in response to local policies and requirements. As a result, individual replicas in the system can respond and adapt to a changing network environment while providing as strong a local guarantee or minimum quality of service as required. The global state of a federated system is defined by the topology of replicas and their interactions, such that if a subset of nodes implement stronger consistency models, then the global probability of conflict is reduced and conversely if a subset of nodes implements weaker consistency then global throughput is increased. Indeed, we find that it is more often the tension between local vs global views of consistency that cause greatest concern in terms of application performance. Because each node can select and change local consistency policies, client applications local to the replica server have greater control of tuning consistency in response to mobile or dynamic network behavior, maximizing timeliness or correctness as needed.

We show that a federated consistency protocol finds a middle ground in the trade-off between performance and consistency, particularly between an eventually consistent system implemented via gossip-based anti-entropy [19] [PJK: HAT...] and a sequential consistency model implemented by the Raft consensus protocol [31]. By exploring these two extremes in the consistency spectrum we show that the overall number of inconsistencies in the system is reduced from the homogenous

eventual system and that the access latency is decreased from the homogenous sequential system. Moreover, because the global consistency of the system is topology-dependent, it can be said to have flexible or dynamic consistency. We have found that large systems with variable latency in different geographic regions can perform well by allowing most nodes to operate in an optimistic fashion, but maintain a strong central quorum to reduce the amount of global conflict.

The rest of the paper is organized as follows: we present a system model that is slightly different from the usual client-server system model and which has many more replica servers participating in a variety of locations. Because we take a client-centric approach we will describe a distributed file system, though our model can be easily generalized to other types of data systems. We will then present a consistency model to describe the impact of conflict, misordering, and staleness and use these metrics to next describe the federated consistency protocol. Finally we will present the performance of our system via experimental simulation and conclude with a discussion of how to adapt this model to include other consistency protocols like causal consistency.

## II. System Model

In order to demonstrate the benefits of federation, we present a system model that shifts away from the traditional cloud-service oriented approach to distributed systems where clients connect to replica servers that may be geographically distributed but usually reside in data centers. We have found that this presentation usually involves many clients connecting to just a few replicas and we intend to discuss replication in medium to large systems with dozens or hundreds of replicas. Instead we present an approach where every client itself is a replica that accesses data locally, which it expects to be both as recent and as correct as possible. To that end we posit a file system as the natural use case of local, client-oriented systems, though we could easily generalize the model to any distributed storage system.

### A. Topology

In order to investigate the effect of variable latency and the network environment on consistency, we have constructed a fully connected topology of replica nodes that are each assigned a geographic region as shown in Figure 1. Within each region, replica nodes enjoy stable, low-latency connections with their neighbors. However, across regions the latency is higher and the connections variable, meaning that out of order messages are more common across the wide area than in the local area.

In this type of topology there are two types of failure: node failure and network partitions. *Node failure* occurs when a single node is shut off or stops responding to messages. *Network partitions* occur when it is not possible for messages to be sent or received from a single geographic region. In both cases, two conditions must be dealt with by the replication protocol in order to satisfy correctness criteria: first the fact that accesses may continue at a partitioned node which are not being replicated by the system and second that the partitioned nodes are behind the global state and must be brought up to date.

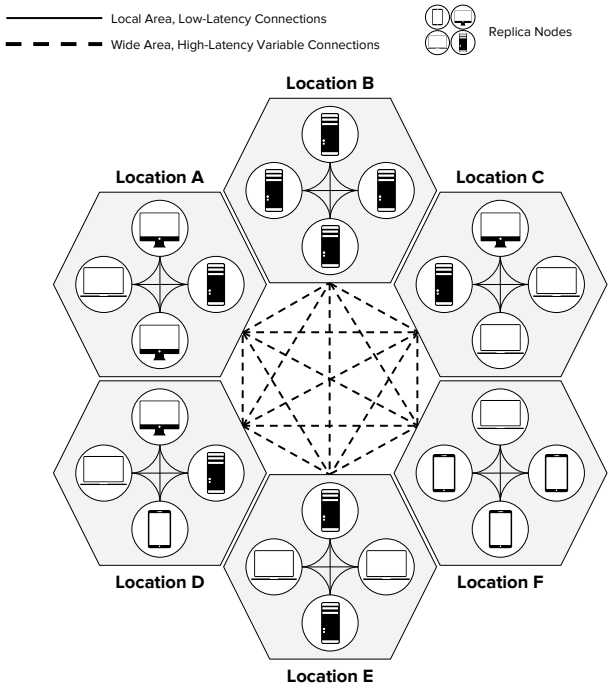


Fig. 1. A topology with many heterogenous replicas operating in a variable latency, partition and failure-prone environment.

### B. Logs and Accesses

Because consistency is defined in terms of an ordering of operations that change the state of a replica, each node maintains a log of operations whose ordering and staleness can be reasoned upon. The question then becomes what those operations are in the context of a file system. A first attempt might be to map all reads and writes as individual operations that must be ordered, however that would be inefficient if the accesses are at the level of buffered system reads and writes. Instead, distributed systems usually aggregate individual accesses into *Close-To-Open* (CTO) consistency where read and write accesses are "whole file" [30]. Furthermore, with respect to local accesses we guarantee that a read returns the last write (given no remote updates, *Read Your Writes Consistency*[PJK: cite]) and that writes are atomic with respect to each other (*Monotonic Write Consistency*) [10].

[Both Read your Writes and Monotonic Write consistency are outlined in [34], [10], [36] with no references; I can report all three above at the end of the sentence?]

Each replica's log therefore is composed of a series of write accesses to multiple objects. Each object has a unique name that identifies it to the system and a monotonically increasing version number which can be implemented either as a vector clock [32] (or a simple Lamport scaler) in the case of a fixed topology or as a vector stamp [4] in the case of dynamic topologies. Therefore a write access encapsulate the following

information: the name of the object being written to, the parent version of the object to which the write is being applied, the versions and object names of any other dependencies, the replica id where the write occurred, , and an array of blob ids that compose the file at the conclusion of the write.

A read access to a particular object simply looks up the latest local version of that object. Because dependency information can be embedded into a write, it is not necessary to include read accesses in the log. For example, in order to create a transaction that reads from objects $X$ and $Y$, performs a computation then updates objects $Y$ and then $Z$: the write to $Y$ would include as a dependency the earlier version of $Y$ and the read version of $X$ and the write to $Z$ would include the updated version of $Y$ and the read version of $X$. Other notions of dependencies include implicit session dependencies, e.g. all writes are dependent on any access that occur within a minimum time threshold of each other, or explicit dependencies that are added by the application.

Because our system model accounts for heterogenous devices and each write in the log is simply metadata about the version of a file we consider version replication as a separate issue from object or blob replication. Furthermore, system consistency depends only on the replication of version information since a version defines what is visible on each replica to be read (and writes follow an implicit read). While a version must become visible (replicated to) all devices in order for the system to be consistent, the blobs that make up data may not be stored on all devices with different storage resources. If a read access requires blobs that it doesn't have, it can simply request them from a local neighbor that does, or from the origin replica itself. For that reason, the rest of this paper will focus on version replication whose primary resource constraint is latency not bandwidth.

## III. Consistency Model

In the previous section, we fixed the client-centric definition of consistency by guaranteeing that accesses should provide at least *Monotonic Writes Consistency* (MTO), which states that two updates by the same client will be serialized in the order they arrive. Because every write maintains its parent version there is an explicit ordering to how writes must be applied. Additionally clients are guaranteed the less strict *Read Your Writes Consistency* (RYWC) such that any accesses after a write to version $n$ will return a version $\geq n$. These client-centric views of consistency ensure that standard file system semantics apply at a local level, though not necessarily globally [10].

Our consistency model is therefore a *data-centric* model which concerns two primary dimensions: ordering and staleness. Given that every replica maintains a local log of accesses with respect to some abstract total ordering based on the sequence of version numbers and potentially concurrent operations, we can define those dimensions as follows:

1) *Ordering* refers to how closely individual logs adhere to the abstract global ordering. A *strict ordering* requires every single log to be exactly the same whereas weaker ordering allows some divergence in the order writes are stored in the log.
2) *Staleness* refers to how far behind the latest global version a local log is and can be either expressed by the visibility latency of replicating a version to all replicas or simply by how many versions the latest is behind by.

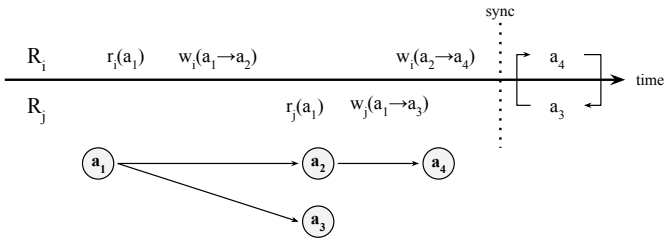


Fig. 2. Accesses before synchronization cause stale reads and forked writes.

Most data-centric consistency models do not consider staleness but instead refer to the strictness of ordering guarantees and the method by which updates are applied to the state of the replica. Indeed, ordering strictness can lead to increased staleness because writes cannot be accepted until they have fulfilled their dependencies first, creating further delay. Our consistency model considers instead the primary symptom of stale reads and writes: *forks*.

**Fork** A fork occurs when two replicas concurrently write a new version to the same parent object. Forks introduce inconsistency because there are now two potential orderings of updates to the log, but forks are primarily the symptom of staleness; e.g. the second writer wrote to a stale version of the object.

All consistency models discussed in this paper with the exception of *linearizability* are susceptible to forks because they allow stale reads to occur. A fork occurs for a single object when two replicas, $i$ and $j$ read object version $A_1$ and then accesses $W_i(A_1)$ and $W_j(A_1)$ occur creating two new versions, $A_2$ and $A_3$ with the same parent version $A_1$ before synchronization can occur as shown in Figure 2. There are two primary causes of forks: concurrent accesses by multiple replicas and stale reads. The former is distinguished from the latter only if the possibility of synchronization could have occurred. From the point of view of the system, they are identical causes. Forks can branch to arbitrary lengths as replicas continue to write to their latest local copies, however when synchronization occurs a decision as to which ordering of writes is correct must occur.

With this context and consistency model in mind, we can now identify several consistency models in order of increasing ordering strictness and define how each model's correctness criteria responds in the face of forks.

### A. Weak Consistency

A weakly consistent system makes no guarantees whatsoever about the relationship of local vs. remote writes and whether or not any given update will become visible [36].

Weak consistency is often described as "replicas might get lucky and become consistent" and in fact a weakly consistent implementation may not have a synchronization protocol whatsoever [10].

Weakly consistent systems may have a role in a Federated environment, however. Consider a simple weak consistency synchronization protocol where any remote updates are accepted so long as they do not conflict with any local updates. This model may be used in the case of a replica that expects to be intermittently offline or unable to perform any conflict resolution. By accepting non-conflicting updates the replica can keep up as much as possible with the rest of the system while protecting a core set of objects that the device will routinely access. For example, in mobile mesh sensor networks, each sensor will be frequently writing to its own objects, updating their state and generally not conflicting with the measurements recorded by other sensors. A weakly consistent system may assists in propagating updates from remote sensors back to a core replica system that implements stronger consistency.

### B. Eventual Consistency

Eventual consistency is primarily concerned with the final state of all logs in the system given some period of quiescence that allows the system to converge. In this case, all replicas, no matter their log ordering, should have identical final versions for all objects in the namespace. This suggests that eventual consistency requires some *anti-entropy* mechanism to propagate writes and a policy to handle convergence [35]. Eventual consistency is very popular for NoSQL databases and hosted distributed storage services [15], [24] because it favors an optimistic approach to consistency: in most systems conflicts are rare, and if something does go wrong, conflict resolution is up to the application layer. In practice, most applications can handle some inconsistency and moreover the small inconsistency windows due to low latencies in cloud data centers make such conflicts rare and short-lived enough to be worth the risk [8].

Eventual consistency implemented by a *last writer wins* policy simply accepts all writes so long as they are more recent than the latest local versions. Eventual reads and writes are always performed locally, and therefore with little performance overhead. Eventual allows forks to occur and moreover allows individual replica logs to have wholly different orderings so long as the last version for each object is in the same given no writes for a long enough period of time. As a result, the latest version of an object may alternate between writes to competing forks (a fairly weak semantic) and in this case, it is up to the application to detect the inconsistency. However, Eventual logs do have one important property - for each object, every write in the log is ordered in a monotonically increasing fashion.

### C. Causal Consistency

Causal consistency provides the strictest ordering guarantee without requiring coordination or consensus [23]. In causal consistency all writes that have causal relationships must have those dependencies satisfied, e.g. inserted into the log before that write can become visible. Therefore, even though a write might have been propagated to another replica server it cannot be read until all of its dependencies have also been propagated. Causal consistency can increase staleness particularly when implicit or potential causality creates large dependency graphs that must be resolved before writes can be applied [29]. This can be managed by allowing the application to explicitly specify the dependencies for each write [6].

Our consistency model provides for causality by ensuring that dependencies are tracked along with specific writes to specific versions of objects; in fact every write has at least one dependency, the parent version of the write. Eventual consistency however does not require that the parent version be appended to the log before the write can be made visible, a requirement for causal. Moreover, parent versions only track causality for a single object and do not consider potential causality or ordering of writes to all objects in the namespace. As a result, in order to maintain inter-object causality each write must explicitly specify its dependencies.

### D. Sequential Consistency

Sequential consistency is a strong consistency model that requires that all replicas have the exact same ordering of their logs, such that all writes by all clients are appended in the same exact order [5]. Sequential consistency is not strict in that it does not make guarantees about staleness (or the ordering of reads) but does require that all writes become visible in the same order [10]. Sequentially consistency is typically implemented with consensus algorithms such as Paxos [25] or Raft [31] that coordinate logs by defining a transitive, global ordering for all conflicts. Alternatively, sequential consistency and can be implemented with warranties – time based assertions about a group of objects that must be met on all replicas before the assertions expired [28].

Because sequential consistency allows stale reads to occur forks are still possible, however only a single branch of a forked write can be allowed in the log. Preventing forks would require either a locking mechanism or an optimistic approach that allowed operations to occur but reject all but one branch. Rejecting writes simply passes the buck back to the application that must deal with dropped writes by either retrying or resolving conflicts and writing a new version.

### E. Linearizability

Linearizability is the strongest form of consistency; not only must all write operations occur in sequence, but all operations including reads must be ordered chronologically [17]. A consensus algorithm alone cannot implement linearizability and instead some distributed locking mechanism is required. For example a consensus algorithm can be adapted to instead of making decisions about the total ordering of conflicting writes, granting or releasing locks from requestors, however this opens up the potential for deadlock and extremely poor performance, defeating the purposes of replication in the first place! Data center environments that don't have to deal with issues of

clock skew by using super precise atomic and GPS clocks can use precise time measurements to enable a distributed two phase commit protocol [14], however every replica is required to have such a time piece, which is not practical for heterogenous topologies.

## IV. Federated Consistency

The Federated Consistency model allows individual replicas to select their own local consistency policies and engage in replication according to the mechanism specified by the policy. Each replica maintains its own local state which is modified in response to local accesses as well as the receipt of messages from remote replicas. Each replica sends messages to other nodes in order to propagate the latest writes as well as to perform housekeeping. Therefore every replica can be seen as an event handler that responds to local access events as well as remote messages and generates more events (sent messages) in return. Simply put, so long as every federated replica has an event handler for all types of RPC messages, federation only has to be defined at the *consistency boundaries*, that is when replicas of one consistency type send messages to that of another.

Given the consistency models discussed in the previous section, we will omit weak consistency as being too simplistic and linearizability as being too performance restrictive. Instead we will focus on the federation of eventual consistency, implemented with latest-writer wins gossip based anti-entropy, and sequential consistency implemented with the Raft consensus algorithm.

### A. Gossip-Based Anti-Entropy

Eventual consistency is implemented by periodic *anti-entropy* sessions that converge replicas towards the same state (e.g. reducing entropy, the divergence between the states of individual replicas) [19]. On a routine interval, specified by the `anti-entropy delay` timing parameter, a replica will randomly select one of the other replicas in the system and send a `Gossip` message that contains the latest version of all objects in the replica's local log. On receipt of the `Gossip` message, the remote replica will compare the RPC object versions with those in its local log. If the RPC versions are later, it will append the later versions of the object to the log (*last-writer wins*). However if the remote object version is later it will send that version back to the originating node in a `GossipResponse` message. As a result, our anti-entropy implementation is *bilateral*.

Replicas that implement eventual consistency read locally and write to their local latest version introducing zero read and write latency. Forks are caused by staleness due to the amount of time it takes to propagate a write to the rest of the system, the visibility latency. Visibility latency in this system can be modeled as:

$$t_{visibility} \approx \frac{T}{4} \log_3 N + \epsilon \tag{1}$$

[PJK: Where does this come from?] [See email – I came up with this for bilateral anti-entropy]

Where $\frac{T}{4}$ is the `anti-entropy delay` as computed from the network environment via the tick parameter, $T$, and $N$ is the number of nodes in the system. The epsilon parameter specifies the amount of added latency injected by imperfect gossip neighbor selections, if $\epsilon = 0$ it would mean that on every anti-entropy session, each node perfectly selected another that didn't have the write being propagated; this is an unlikely scenario in a system that selects neighbors based on uniform random probabilities.

### B. Raft Consensus

Sequential consistency is implemented via the Raft consensus algorithm [31]. However, consensus alone does not ensure that sequential consistency is implemented and a number of policy decisions about how Raft followers read and write and interact with the leader must be discussed. First, we will present a brief sketch of the Raft consensus algorithm.

All Raft nodes can be in one of three states: `FOLLOWER`, `CANDIDATE`, and `LEADER` and all replicas are initialized in the `FOLLOWER` state. Raft is governed by two primary timing parameters: the `heartbeat interval`, which specifies how often the leader sends `AppendEntries` messages that double as term keep-alive messages, and the `election timeout`, an interval in which a uniform random delay is chosen if by which a follower doesn't hear from the leader, it will convert to a candidate and attempt to gain enough votes to become a leader. After initialization, one of the replicas will timeout waiting for an `AppendEntires` and will send `VoteRequest` messages with a monotonically increasing term id. Replicas will vote for the candidate if and only if the term is greater than their term and if they haven't voted for anyone in the current term. Once a candidate receives a majority of votes from other replicas it becomes the leader.

The Raft leader has the primary responsibility of coordinating all other Raft replicas. To that end, the leader will broadcast periodic `AppendEntries` messages to all other Raft followers in order to maintain their leadership for the given term. A write access that originates at a follower must be sent as a `RemoteWrite` to the leader. The leader accepts writes in the order that they are received, and if the leader detects a fork – that is that a write has a parent version who already has a child version in the log – Raft will simply reject (drop) the write. In order to minimize the number of messages that Raft sends, Raft will aggregate all writes into the next `AppendEntries` message and send them together.

Because all writes that originate at followers are forwarded to the leader, the leader can guarantee a sequential ordering of updates. Therefore on receipt of an `AppendEntries` message, followers simply add the entries to their log and respond with their last index. If a majority of followers append entries to their logs, the leader will mark those entries as committed and inform the followers the write has been committed on the next `AppendEntries`.

However, although all writes are sequentially ordered, Raft nodes must decide what to do read, and there are several options:

1) *READ COMMITTED* Raft replicas will only read the latest committed version of an object, guaranteeing that the write will not be rolled back in the case of an outage. However, this read mode introduces the potential for a lot of staleness and therefore forks.
2) *READ LATEST* Raft replicas will read the latest version of the object in their log, even if it hasn't been committed. Moreover, replicas will read their own local writes rather than waiting for an `AppendEntries` to return their write.
3) *REMOTE READ* Rather than read locally, simply request the latest version from the leader. This introduces the potential for additional latency, but may be faster if the expected message latency is less than the heartbeat interval.

Each of these options has critical implications for the likelihood of stale reads and writes in the system. Replicas would choose read committed if the network was highly partition prone and messages from the leader were unstable and prone to being rolled back. Remote read servers replicas well when the average message latency is far lower than the heartbeat interval, though this could be improved by making the heartbeat interval similar to the network latency. For this reason, we have selected read latest as the most likely scenario for a file system implementing sequential consistency with Raft.

### C. Timing Parameters

Both the eventual and sequential consistency models are hugely dependent on the timing parameters for consistency. In order to select an `anti-entropy delay`, `heartbeat interval`, and `election timeout` we must find some method of relating the timing to the base latency of our system. To do this we introduce a "tick" parameter, $T$, that is computed by the observed message latency in the system specified as normally distributed with a mean, $\lambda_\mu$ and standard deviation, $\lambda_\sigma$. There are two formulations of T: a conservative formulation that is big enough to withstand most variability, and an optimistic formulation that is much faster but will send many out of order messages that can disrupt consistency if there is variability.

$$T_{conservative} = 6(\lambda_\mu + 4\lambda_\sigma) \tag{2}$$

$$T_{optimistic} = 2(\lambda_\mu + 4\lambda_\sigma) \tag{3}$$

Note that both $T_{conservative}$ and $T_{optimistic}$ are variations on other known models. Ongaro and Ousterhout proposes a more conservative $T$ parameter of $10\lambda_\mu$ [31], which while correct has very poor performance relative to the access stream. Howard et. al proposes a similar optimistic $T$ parameter, $\lambda_\mu + 2\lambda_\sigma$ [18] – but this does not capture enough of the variability in the networks we propose and leads to out of order `AppendEntries` messages, which in turn leads to message thrashing and can impair Raft by causing the log to be entirely replayed.

With the $T$ parameter, all timing parameters for all consistency models can be specified in relationship with each other. For example, in order to ensure that eventual and sequential replicas send approximately the same number of messages (e.g. fixing the message budget in resource constrained environments) the timing parameters are as follows:

- `anti-entropy delay` $= \frac{T}{4}$
- `heartbeat interval` $= \frac{T}{2}$
- `election timeout` $= U(T, 2T)$

Moreover, $T$ can be adapted at runtime. For eventual nodes, $T$ can be continually updated with respect to observed message latencies. For Raft nodes, the leader can observe message latencies in response to `AppendEntries` messages and initiate joint consensus in order to change the configuration. Once joint consensus is achieved the new $T$ tick parameter can be applied.

### D. Protocol Federation

[Write a description of how federation is implemented and the "silver bullet" to coordinate between eventual and Raft. The following section is excerpted from the proposal.]

A key requirement of Federated Consistency is the opportunity to create heterogeneous systems with no performance cost, e.g. a homogenous eventual cloud and a homogenous Raft cloud will continue to perform equivalently whether or not they participate in a federated cloud. However, we posit that an eventual cloud should benefit in lowered data staleness and in fork frequency from being connected to a strong, central consensus group. Similarly, Raft nodes should be able to use anti-entropy mechanisms to replicate data and continue writing even if the leader is unavailable and no consensus can be reached to elect a leader. The question is therefore how to integrate the eventual consistency via gossip and sequential consistency via consensus in a non-invasive way.

The central problem is that eventual and Raft clouds choose the "winner" of a fork in exactly opposite ways. Eventual clouds choose the last of a set of conflicting writes through a latest-writer wins policy, whereas Raft clouds effectively choose the first by dropping any write that conflicts with any previously seen writes. Our insight is that if the strong central quorum can somehow make an accepted version "more recent" than a dropped fork, then the propagation of the fork would cease in the eventual cloud, reducing the possibility of continued forks.

In order to federated multiple consistency models, there are two integration points: communication and consistency. Our initial approach was to integrate communication at the Raft nodes, by allowing Raft nodes to participate in anti-entropy with the eventual cloud (but not other Raft nodes). Eventual nodes therefore "synchronize" with a local Raft node (modified by some synchronization probability) by exchanging `Gossip` messages with the Raft nodes. A slight increase in the synchronization probability balanced the amount of synchronization with the amount of communication in the eventual cloud given the imbalance in the ratio of eventual

nodes to Raft nodes. In order to manage the communications delay between the anti-entropy timeout and leadership coordination, Raft nodes must keep local caches of forked or dropped writes so as to not propagate them back to the eventual cloud or replay them to the leader. However, we have observed that this was not enough to stop the eventual cloud from propagating a fork around Raft, causing further inconsistencies.

Our approach to integrate consistency is to extend each version number with an additional monotonically increasing counter called the *forte* (strong) version that can only be incremented by the leader of the Raft quorum. Because the Raft leader dropped forks or any version that was not more recent than the latest version, incrementing the forte number on commit ensures that only consistent versions are marked. In order to determine the latest version, the forte number is compared first, then the version number, allowing Raft to "bump" the consistent version to a more recent version. In order to prevent that version's children from becoming less recent, on receipt of a version with a higher forte than the local, eventual nodes must search for the forte entry in their local log, find all children of the update, and set the child version's forte equal to that of the parent.

Our preliminary investigations show that this integration works well to curb inconsistencies due to anti-entropy delays as well as those introduced by communication integration. There are, however, quite a few knobs to turn in the system described above. Further investigation into smoothing integration points between consistency protocols and the policies that each define may lead to smoother messaging with fewer resource constraints. We have identified a current bottleneck in the system however: the leadership of the central quorum, through which every single write must pass, no matter the size of the cloud. In order to address this, we propose an adaptation to the central consensus group such that it can provide Hierarchical Consensus.

## V. Simulation Results

In order to evaluate a federated model of replication and consistency, we have created a discrete event network simulation that allows us to flexibly configure a variety of parameters and more closely investigate the behavior of replication. The input to the simulation has two parts: a topology that specifies the replicas and network environment; and a workload of access events to named objects. The simulation instantiates each replica as a process that executes read and write accesses to objects and generates replication messages as well as listens for and handles messages from other replica servers.

Each topology specifies each device as an independent replica server, by uniquely identifying it with device-specific configurations. By far the most important configuration is the *replica consistency* (or replica protocol) which determines the replica's behavior in the environment. For the purposes of our simulation we have proposed three types of eventual and sequential consistency implementations:

- **Eventual**: Eventual nodes perform replication via anti-entropy by random selection of a neighbor to do pairwise gossip with. Eventual nodes can prioritize local vs. wide area nodes by specifying $P_{local}$ – the likelihood of local neighbor selection.
- **Stentor**: A "Stentor" node is an eventual node that performs *two* anti-entropy sessions per interval, always choosing one local and one remote node. We have implemented Stentor nodes to show that Raft isn't simply acting as a minimum spanning tree in the network, but is also preserving consistency guarantees.
- **Raft**: Raft nodes implement the Raft consensus protocol, electing a leader and forwarding writes to the leader to maintain a sequential ordering. If a write is forked, the leader will drop it in order to maintain our consistency guarantee.

The topology further specifies the *location* of each device, the *connections* between devices, and the *distribution* of message latency on a per-connection basis. There were two primary latencies specified as normal distributions ($\lambda_\mu$, $\lambda_\sigma$), the *local area* latency, usually with a lower $\lambda_\mu$ and $\lambda_\sigma$ than the *wide area* latency – e.g. the latency between devices in different locations. In order to compute the tick parameter, $T$, and specify the average latency in the simulation, latencies are given as the worst case, wide-area latencies. Each topology could also set simulation-specific configurations, as well as device-specific configurations.

The workload was specified as access trace files – time ordered access events (reads and writes) between a specific device and a specific object name. Each trace was constructed via a random workload generator, where a collection of available devices was specified along with a normal distribution of the delay between accesses ($A_\mu$, $A_\sigma$), the number of objects, $o$, in the system, and a probability of conflict, $P_c$. To generate the workload, object names were assigned to each replica as follows: in a round-robin fashion, an object name was selected and assigned to a replica with probability $P_c$ until each replica was accessing $o$ objects. If $P_c = 1.0$ then every single replica would be accessing the same batch of objects, where as if $P_c = 0.0$ then each replica would access their own unique set of objects. From there, the $A_\mu$, $A_\sigma$ was used to generate accesses to objects in sequence, by selecting an object and reading and writing to it over time until some probability of switching objects occurred. In effect, the final workload simulates multiple replicas reading and writing at a moderate pace for approximately one hour.

Examples of topologies and accesses are shown in Figures 3 and 4; these graphs show our primary simulation topology of 20 nodes, 4 nodes each in 5 wide areas, the size of vertices represents the number of accesses that occur at that location and the color represents the replica type, the edges are colored by RPC type and sized by the number of messages. These figures show that minor changes in the topological configuration can lead to dramatic changes in the message traffic in the system; both of these figures show a federated topology of a small Raft core group surrounded by eventual

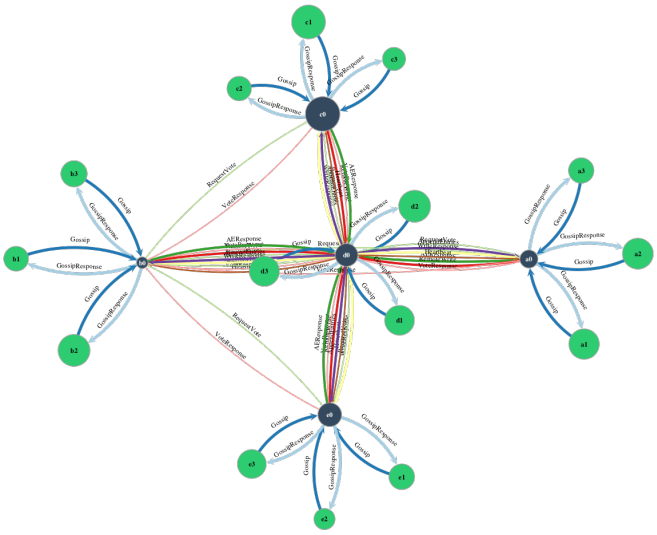

Fig. 3. Accesses in a Federated topology with primary Raft synchronization.

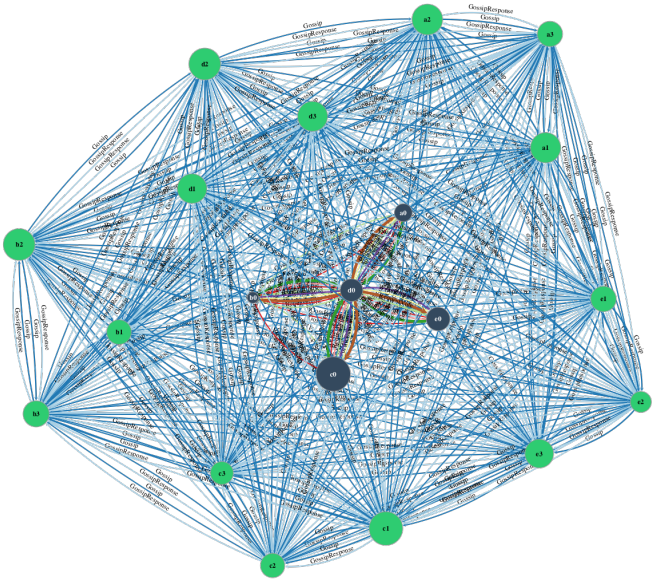

Fig. 4. Accesses in a Federated topology with a preference for wide area anti-entropy over synchronization.

clients however Figure 4 has a preference for anti-entropy across the wide area whereas Figure 3 has a preference to synchronizing with the local Raft node.

### A. Experiments and Metrics

We conducted two primary experiments to test the behavior of a federated consistency system against homogenous Raft and homogenous Eventual systems. The first was to test the behavior of increasing conflict, $P_c$ and the second was to explore the effect of the network environment in terms of the mean latency, $\lambda\mu$ of the wide area. The input to the system was a fixed topology of 20 nodes in 5 wide areas, along with an access trace that contained approximately 29,000 accesses (depending on the experiment), with an average ratio of 54% of the accesses being reads [PJK: Why 54%? More generally you need to motivate our choices for trace and topology.].

Our primary metrics are *forked writes* and *stale reads*. We define forked writes as the number of writes that had more than one child, whereas a read is stale if it returns anything other than the globally latest version. A further metric of *inconsistent writes* measured the number of writes whose parent was forked allowed into the log of a replica, as balanced by the number of *dropped writes* — writes that the sequential ordering Raft nodes would reject as being invalid.[PJK: Motivate all.]

Furthermore the cost of each replication protocol was measured in terms of the number of *sent messages* as well as latencies of various RPC calls. Some latencies were dependent if the protocol specified *cached* or *remote* accesses: the *read latency* and *write latency* measured the amount of time an access took until a response was returned to the client. In the case of Eventual nodes, this latency was zero, since an eventual node always responds with cached information; however Raft required remote accesses to the leader in order to respond. Other latencies were global, e.g. the *visibility latency* specifies the average amount of time it takes a write to become fully visible, that is replicated to all nodes in the network. Any writes that do not become fully visible (stomped on through the eventually consistent policy of latest writer wins) are ignored. This metric is closely related to the *percent visible* metric — the average number of replicas a write is propagated to. Finally the *commit latency* specified how much time was required for a write to be committed, e.g. confirmed as accepted by a quorum of nodes.

### B. Conflict Probability

In this experiment we ran 44 simulations, each of which specified a combination of consistency type: one of *homogenous eventual*, *homogenous raft*, *federated eventual with 5 raft nodes*, and *federated eventual with 5 stentor nodes* as well as a $P_c \in [0.0, 1.0]$ with a step of $0.1$. In particular, we wanted to show how a Federated system would behave in the best case scenario for all replication protocols (no conflict, each replica accesses only their own objects) to the point where conflict is guaranteed because all replicas are trying to read and write to the same small set of objects.

As shown in Figure 5, as the conflict in the system increases, eventual methodologies suffer from more and more inconsistent writes[PJK: And we care because...]. Eventual nodes do manage forks or ordering of accesses[PJK: huh?], but simply write to the latest version. This has benefits in that eventual

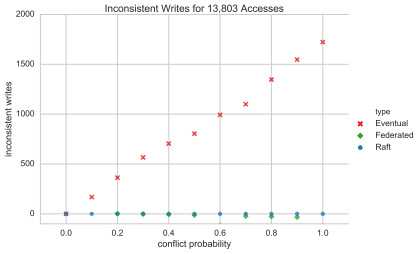


Fig. 5. Inconsistent writes by increasing conflict.

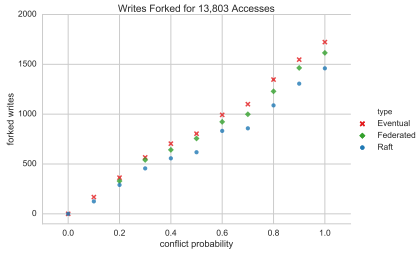


Fig. 6. Forked writes caused by increasing conflict.

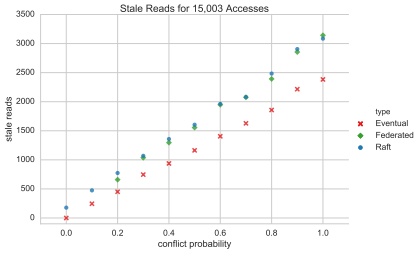


Fig. 7. Stale reads as conflict increases.

nodes can respond quickly, and in fact as shown in 7, eventual systemshave low numbers of stale reads even as conflict increases. However, as shown in 6, the inconsistencies that are in the eventual nodes match the number of forked writes in the system[PJK: so?]. However, both Raft and Federated systems eliminate inconsistencies (Federated allows a few in higher conflict scenarios) by simply dropping writes. [PJK: Why is this important? User view of data more orderly?]

### C. Latency Variation

In the second experiment we ran 216 experiments to test the effect of variable network latency on consistency protocols as well as on how the selection of the tick parameter model affects consistency for each system. We specified 12 wide area mean latencies in three categories: low ($\lambda\mu \in [40, 500]ms$ with a low $\lambda_\sigma = 10.0$), medium ($\lambda\mu \in [500, 1000]ms$ with a moderate $\lambda_\sigma = 20.0$), and high ($\lambda\mu \in [1000, 1500]ms$ with a high $\lambda_\sigma = 30.0$). In each of these environments we ran three replication models: *homogenous eventual*, *homogenous Raft*, and *federated eventual and Raft*.

Each of these 108 simulations was then parameterized with two different $T$ parameters, computed by the wide area $\lambda_\mu$ and $\lambda_\sigma$: $T_{conservative}$ and $T_{optimistic}$. However the access mean, $A_\mu$ was fixed at approximately one access per replica every 3000ms. In effect, this meant that for approximately half the simulations (with the higher latencies), it was impossible for a write to become visible on another replica before a fork. Even though we fixed the conflict probability as $P_c = 0.5$ there was still enough conflict in the simulation that forced each replica protocol to handle forks.

In terms of consistency metrics, the number of inconsistent writes[PJK: Again, this is a low-level metric that affects other metrics (forks, staleness) that are application/programmer visible. We might want to not even mention IWs except to explain strange results.] is more than halved from eventual in Federated due to the presence of the Raft core (Figure 10). This is because Federated can take advantage of the topology where Raft and Eventual cannot - broadcasting across the wide area to minimize the number of pairwise communications in anti-entropy but still allowing available responses with the Eventual core. Similarly, the number of stale reads as shown in Figure 12 sits between both Raft and Eventual for both the optimistic and conservative tick parameters.

We can characterize this in two ways: first that the propagation is faster in Federated than it is in Eventual, thereby leading to fewer forks. Figure 9 shows the average visibility latency, that is the time in milliseconds it takes a version to fully replicate, that is become visible on every node in the network. Raft benefits from a broadcast mechanism, while Eventual must wait for exponential propagation. Federated is somewhat slower than Raft, but outperforms Eventual while still being fault tolerant and highly available. As a result, as shown in Figure 8, the number of fullly visible writes is higher for Federated than Eventual, since there is less stomping on versions (maintained by Raft) and because of the decrease in the visibility latency. Together these two properties give Federated better consistency than Eventual, while still maintaining availability in Raft. [PJK: I'd say that Fig 11+12 are the core of the paper, and they are quite poor for federated. Please don't forget to get the cumulative staleness numbers.]

## VI. Discussion

A strong central core to provide support to the entire system has been suggested both in Oceanstore [22] and primary copy schemes [16]. We take this idea further in our experiments by Federating a strong central core composed of Replicas that perform consensus via the Raft consensus protocol and combine it with highly available eventually consistent systems. In this way Federated consistency gains the flexibility and availability of the eventual nodes (leader re-election and no requirement for remote writes mean that the node can continue even if it is completely partitioned from the rest of the network) while still getting guarantees from Raft, which minimizes "fork flipping" – the behavior of writing to one branch then another, truly pernicious inconsistent behavior that cannot be prevented in an eventually consistent system.

System designers can take advantage of heterogenous nodes by implementing stronger consistency on more reliable machines that are able to handle more messages. Mobile nodes that are prone to network loss, out of order or missed messages, or other variable behavior can adapt their policy depending on the environment they're in. Our model also allows for *adaptive* behavior, in that the replicas can monitor the environment for change and as the mean latency decreases, adapt their $T$ parameter accordingly. This is a no cost operation for eventual nodes (who can also optimize pairwise gossip by implementing non-discrete random selection using

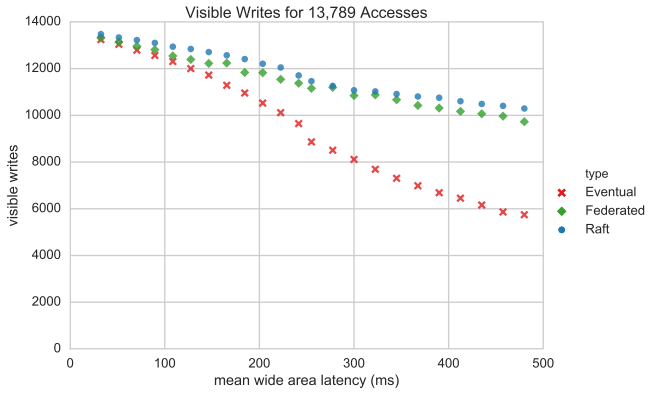


Fig. 8. The percentage of fully visible writes.

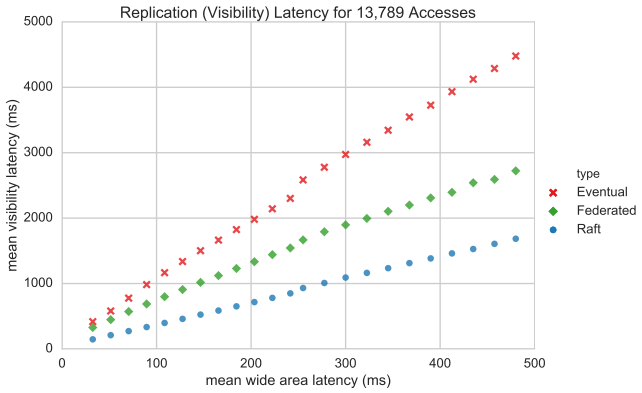


Fig. 9. The average full visibility latency.

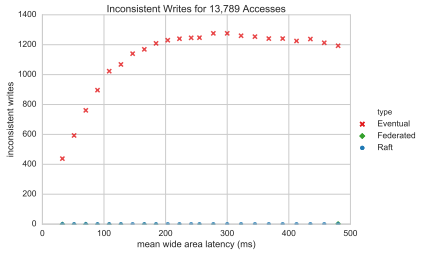


Fig. 10. The number of inconsistent writes written to the log of a replica.

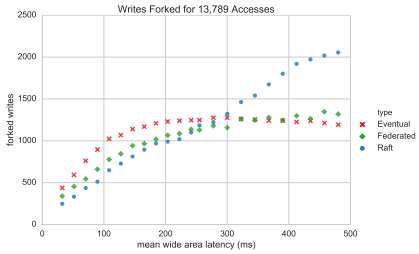


Fig. 11. The total number of conflicts (possible forks).

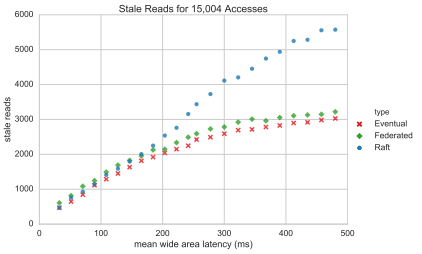


Fig. 12. The percent of reads that are stale in the system.

Bandits or other optimization techniques), and only requires joint consensus on the part of the consensus group.

Our simulation implementation gave us the benefit of being able to manipulate many parameters to see how replication behavior changes. As a result, we only focused on the number of consistency messages sent. Potentially this overstates the differences between the protocols; Raft leaders have more processing requirements for example. Additionally bandwidth pressure and differences in messaging might be more important to the overall performance of a system. However, by focusing on the number of messages and isolating version replication from blob replication (e.g. minimizing the amount of data required for consistent behavior), we have allowed each protocol to fix some "message budget".

In truly resource constrained environments, allowing both Raft and Eventual (either in a homogenous or Federated context) to maintain an equal or semi-equal message budget by modifying the $T$ parameter and allocating timing similarly means that congestion is reduced or optimized. Moreover, this system also allows for adaptive behavior by providing a single place of modification and optimization; $T$ could be optimized according to a cost function (e.g. the message budget or other real time estimates of performance) or could be boosted for a node that has to do a lot of work. This type of flexibility is important to being able to understand how minor changes in both the environment and protocols effect overall consistency in the system as a whole.

[PJK: Given bilateral anti-entropy and a large communication budge, nothing beats eventual. In some situations Raft might be "better" because it's consensus, but even w/ it we can't implement sequential consistency. So where is raft better? Everyone sees the current version at the end of an epoch. Note that raft doesn't prevent stale reads, but it does bound the staleness (to the length of an epoch). Eventual doesn't. W/ federated, we lose the ability to bound staleness tightly, so we need to make sure that we implement out epoch-backpressure. I'm hopeful the resulting numbers will look much better.]

[Since the note above, we've implemented the forte number (backpressure)]

### A. *Fork Minimization*

The primary form of inconsistency that we are concerned with is a fork, that is two separate, conflicting writes to the same parent version. In a homogenous eventually consistent system implemented with anti-entropy, the only way to minimize the number of forks is by increasing the speed of propagation. We have presented the best case for eventual by implementing *bilateral* anti-entropy as well as neighbor selection likelihoods that lead to exponential convergence of a write. However, with a *latest-writer wins* policy, the possibility of "branch-flopping" exists; that is after a fork occurs, the latest version is updated on either side of the branch, and though the system will converge to a final write, branches

may be arbitrarily long and represent significantly inconsistent states to the user.

A homogenous Raft system on the other hand simply does not allow forks to occur by requiring that the leader determines which write is accepted and which is not. Because we are aggregating writes in the `AppendEntries` messages every heartbeat interval, at most $n-1$ forks, where $n$ is the number of nodes, are possible within the heartbeat interval. Whichever remote write arrives at the leader first will be accepted as the canonical branch. Here, the branch length is bounded to the number of writes possible by a single node in the heartbeat interval, which is small – and the branch is eliminated at the heartbeat interval anyway.

The Federated system must at its core resolve how fork elimination is communicated to the eventual system. A fork, by definition, is later than the earlier version. If the Federated system simply ignores or drops forks, the eventual cloud might still propagate the fork around Raft. Worse, if the eventual portion simply drops forks without deciding on a write to propagate, the system is no longer eventually consistent since it is possible that given no more writes half of the nodes make one fork visible while the other half make the other visible.

**Note: right now we have Federated simply dropping the fork, which is creating bushier version trees and therefore more dropped writes.**

## VII. Related Work

One of the earliest attempts to hybridize weak and strong consistency was a model for parallel programming on shared memory systems by Agrawal et al [2]. This model allowed programmers to relax strong consistency in certain contexts with causal memory or pipelined random access in order to improve parallel performance of applications. Per-operation consistency was extended to distributed storage by the RedBlue consistency model of Li et al [26]. Here, replication operations are broken down into small, commutative suboperations that are classified as red (must be executed in the same order on all replicas) or blue (execution order can vary from site to site), so long as the dependencies of each suboperation are maintained. The consistency model is therefore global, specified by the red/blue ordering and can be adapted by redefining the ratio of red to blue operations, e.g. all blue operations is an eventually consistent system and all red is sequential.

The next level above per-operation consistency hybridization is called *consistency rationing* wherein individual objects or groups of objects have different consistency levels applied to them to create a global quality of service guarantee. Kraska et al. [20] initially proposed consistency rationing be on a per-transaction basis by classifying objects in three tiers: eventual, adaptable, and linearizable. Objects in the first and last groups were automatically assigned transaction semantics that maintained that level of consistency; however objects assigned the adaptable categorization had their consistency policies switched at runtime based on a cost function that either minimized time or write costs depending on user preference. This allowed consistency in the adaptable tier to be flexible and responsive to usage.

Chihoub et al. extended the idea of consistency rationing and proposed limiting the number of stale reads or the automatic minimization of some consistency cost metric by using reporting and consistency levels already established in existing databases [12], [13]. Here multiple consistency levels are being utilized, but only one consistency model is employed at any given time for all objects, relaxing or strengthening depending on observed costs. By utilizing all possible consistency semantics in the database, this model allows a greater spectrum of consistency guarantees that adapt at runtime.

Al-Ekram and Holt [3] propose a middleware based scheme to allow multiple consistency models in a single distributed storage system. They identify a similar range of consistency models, but use a middleware layer to forward client requests to an available replica that maintains consistency at the lowest required criteria by the client. However, although their work can be extended to deploying several consistency models in one system, they still expect a homogenous consistency model that can be swapped out on demand as client requirements change. Additionally their view of the ordering of updates of a system is from one versioned state to another and they apply their consistency reasoning to the divergence of a local replica's state version and the global version. Similar to SUNDR, proposed by Li et al. [27], an inconsistency is a fork in the global ordering of reads and writes (a "history fork"). Our consistency model instead considers object forks, a more granular level that allows concurrent access to different objects without conflict while still ensuring that no history forks can happen.

Hybridization and adaptation build upon previous work that strictly categorizes different consistency schemes. An alternative approach is to view consistency along a continuous scale with a variety of axes that can be tuned precisely. Yu and Vahdat [37] propose the *conit*, a consistency unit described as a three dimensional vector that describes tolerable deviations from linearizability along staleness, order error, and numeric ordering. Similarly, Afek et al. [1] present quasi-linearizable histories which specify a bound on the relative movement of ordered items in a log which make it legally sequential.

## VIII. Conclusion

In this paper we have presented a model for federated consistency that allows individual replicas to expose local policies to users while still allowing for global guarantees given a heterogenous system of replicas where at least a core group enforces sequential consistency. By designing a federated system where only the interactions between replicas of varying consistency types are defined, systems can scale beyond the handful of devices usually described to dozens or hundreds of replicas in variable latency, partition-prone geographic networks. As each replica monitors its local environment and the throughput of its communication with

other replicas it can adapt as necessary to the timeliness vs. correctness constraints required by the local user.

## ACKNOWLEDGMENTS

Thank you Bluejacket, for tirelessly running simulations as soon as we spun you up.

## REFERENCES

[1] Yehuda Afek, Guy Korland, and Eitan Yanovsky. Quasi-linearizability: Relaxed consistency for improved concurrency. In *International Conference on Principles of Distributed Systems*, pages 395–410. Springer, 2010.

[2] Divyakant Agrawal, Manhoi Choy, Hong Va Leong, and Ambuj K. Singh. Mixed consistency: A model for parallel programming. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 101–110. ACM, 1994.

[3] Raihan Al-Ekram and Ric Holt. Multi-consistency data replication. In *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*, pages 568–577. IEEE, 2010.

[4] Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. Version stamps-decentralized version vectors. In *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, pages 544–551. IEEE, 2002.

[5] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems (TOCS)*, 12(2):91–122, 1994.

[6] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. The potential dangers of causal consistency and an explicit solution. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 22. ACM, 2012.

[7] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *Proceedings of the VLDB Endowment*, 5(8):776–787, 2012.

[8] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Quantifying eventual consistency with PBS. *The VLDB Journal*, 23(2):279–302, 2014.

[9] David Bermbach, Markus Klems, Stefan Tai, and Michael Menzel. Metastorage: A federated cloud storage system to manage consistency-latency tradeoffs. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 452–459. IEEE, 2011.

[10] David Bermbach and Jörn Kuhlenkamp. Consistency in distributed storage systems. In *Networked Systems*, pages 175–189. Springer, 2013.

[11] U. Cetintemel, P. J. Keleher, B. Bhattacharjee, and M. J. Franklin. Deno: A Decentralized, Peer-to-Peer Object Replication System for Mobile and Weakly-Connected Environments. *IEEE Transactions on Computers (TOC)*, 52(7), July 2003.

[12] Houssem-Eddine Chihoub, Shadi Ibrahim, Gabriel Antoniu, and Maria S. Perez. Harmony: Towards automated self-adaptive consistency in cloud storage. In *2012 IEEE International Conference on Cluster Computing*, pages 293–301. IEEE, 2012.

[13] Houssem-Eddine Chihoub, Shadi Ibrahim, Gabriel Antoniu, and Maria S. Perez. Consistency in the cloud: When money does matter! In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 352–359. IEEE, 2013.

[14] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, and others. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[15] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[16] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *ACM SIGMOD Record*, volume 25, pages 173–182. ACM, 1996.

[17] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[18] Heidi Howard, Malte Schwarzkopf, Anil Madhavapeddy, and Jon Crowcroft. Raft refloated: Do we have consensus? *ACM SIGOPS Operating Systems Review*, 49(1):12–21, 2015.

[19] David Kempe, Alin Dobra, and Johannes Gehrke. Gossip-based computation of aggregate information. In *Foundations of Computer Science, 2003. Proceedings. 44th Annual IEEE Symposium on*, pages 482–491. IEEE, 2003.

[20] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. Consistency rationing in the cloud: Pay only when it matters. *Proceedings of the VLDB Endowment*, 2(1):253–264, 2009.

[21] Sudha Krishnamurthy, William H. Sanders, and Michel Cukier. An adaptive framework for tunable consistency and timeliness using replication. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 17–26. IEEE, 2002.

[22] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, and others. Oceanstore: An architecture for global-scale persistent storage. *ACM Sigplan Notices*, 35(11):190–201, 2000.

[23] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems (TOCS)*, 10(4):360–391, 1992.

[24] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[25] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[26] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Presented as Part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 265–278, 2012.

[27] Jinyuan Li, Maxwell N. Krohn, David Mazieres, and Dennis Shasha. Secure untrusted data repository (SUNDR). In *OSDI*, volume 4, pages 9–9, 2004.

[28] Jed Liu, Tom Magrino, Owen Arden, Michael D. George, and Andrew C. Myers. Warranties for faster strong consistency. *Proc. of the 11th USENIX NSDI*, 2014.

[29] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416. ACM, 2011.

[30] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 174–187. ACM, 2001.

[31] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, 2014.

[32] D. Stott Parker, Gerald J. Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David Edwards, Stephen Kiser, and Charles Kline. Detection of mutual inconsistency in distributed systems. *IEEE transactions on Software Engineering*, (3):240–247, 1983.

[33] Evaggelia Pitoura and Bharat Bhargava. Data consistency in intermittently connected distributed systems. *IEEE Transactions on knowledge and data engineering*, 11(6):896–915, 1999.

[34] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. Session guarantees for weakly consistent replicated data. In *Parallel and Distributed Information Systems, 1994., Proceedings of the Third International Conference on*, pages 140–149. IEEE, 1994.

[35] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. *ACM*, 29, 1995.

[36] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[37] Haifeng Yu and Amin Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Transactions on Computer Systems (TOCS)*, 20(3):239–282, 2002.